This general-purpose architecture speeds up computationally intensive tasks by augmenting the core processor's functionality with new operations. The operations are synthesized from information extracted during compilation.

General-purpose computers are designed with the primary goal of providing acceptable performance on a wide variety of tasks rather than high performance on specific tasks. The performance of these machines ultimately depends on how well the capabilities of the processing platform match the computational characteristics of the applications. If an application requires more computational power than a general-purpose platform can achieve, users are often driven to an application-specific computer architecture in which fundamental machine capabilities are designed for a particular class of algorithms. Tasks suited to a given application-specific machine perform well, but tasks outside the targeted class usually perform poorly.

Computationally intensive applications typically spend most of their execution time within a small portion of the executable code. A general-purpose machine can substantially improve its performance in many of these applications by adapting the processor's configuration and fundamental operations to these frequently accessed portions of code. Segments of the processing platform can be reconfigured to add new capabilities that customize the architecture to individual tasks. Such an architecture retains its general-purpose nature, while reaping the performance benefits of application-specific architectures.

In this article, we review some of the issues in adaptive computing systems and describe the architecture and compiler components of a general-purpose computing platform called PRISM (Processor Reconfiguration through Instruction-Set Metamorphism). We also describe PRISM-I, an initial prototype system, and present experimental results that demonstrate the benefits of the PRISM concept.

**Adaptation in computing systems**

The idea of incorporating some means of adaptation into a computer has been around for almost as long as the digital computer itself (see the sidebar "Early work in adaptive architectures"). Recent studies have demonstrated the feasibility and
Early work in adaptive architectures

In 1960, Estrin proposed a machine architecture consisting of a fixed, general-purpose core appended with an inventory of high-speed substructures. The fixed portion of the computer centered on a “minimum vocabulary and machine characteristics” common to many applications. A rack of substructures, or application-specific computational elements, provided the means of tuning the fixed core towards application-specific tasks. Unfortunately, the substructure library would have offered only a small number of alternatives from which to choose. Furthermore, the high-level-language compiler would have been easily overwhelmed with the task of matching individual library functions with program behavior.

Later, in the 1960s and 1970s, many large machines were built with writable control stores. These machines required substantial microcoding and decoding to implement their large, complex instruction sets. Nevertheless, a writable control store offered a way of utilizing the faster access time of RAM structures; it also allowed easy microcode maintenance of machines in the field. In 1978, Rauscher and Agrawala introduced a method of task adaptation in machines with writable control stores. This method used information derived from the user’s program to generate new machine microcode at compile time. Their proposed compiler defined a new set of instructions better suited for the task at hand and generated microprograms that would interpret the new instructions.

By the late 1980s, memory technology had advanced to the point where entire programs could be compiled into what was essentially “microcode” resident in the machine’s fast store. The RISC machines resulting from this technology rendered many of the earlier large microcoded machines obsolete.

References


The PRISM approach

A configuration compiler is specialized to accept a program as input and
Compiling a hardware implementation

Compiling a high-level-language specification into an efficient hardware implementation is a difficult problem. One reason is that the programmer is limited to the fundamental operations inherent in the high-level language to convey an algorithm's concepts. All functions, no matter how complicated, must be expressed as a combination of the language's set of Boolean and arithmetic operations.

However, not all functions are easily expressed in these simple operations. For example, Figure A shows a function Hamming written in C. As defined here, this function accepts two unsigned integers and returns a number from zero to 16 equal to their Hamming distance. The C language includes no primitive to perform the Hamming metric. Thus, expressing it in the C functional set, as in Figure A, becomes an expensive operation as compared to say, integer addition. Applications that depend heavily on such a function would obviously benefit if the underlying processor supported it and the high-level-language compiler could use it.

There are a number of ways to express this function; depending on several factors, some ways may be computationally more efficient than others. Independent of how the function is expressed, the most efficient method of execution is direct hardware evaluation. However, manufacturers of general-purpose microprocessors do not ordinarily include instructions such as Hamming functions in their repertoire. The rare occurrence of these instructions in a general-purpose instruction mix does not justify the allocation of silicon area to them. Instead, manufacturers strive for instruction sets that are complete in the sense that the instructions not implemented in the set can at least be synthesized from those that are.

```
short
Hamming(11, 12)
unsigned short i1, i2;

// Returns the Hamming distance of i1 and i2.
// The Hamming distance is a metric determined
// by the sum of the absolute bitwise difference
// of the two operands.

register short i, /* loop counter */
xor, /* intermediate XOR */
result; /* final return value. */

result = 0;
xor = i1 ^ i2;
for (i = 0; i < 16; i++) {
result = (xor & 1) + result;
xor = xor >> 1;
}
return( result );
```

Figure A. The Hamming metric written in C.

Reconfigurable media will always have slower propagation times and lower effective gate densities than a counterpart ASIC (application-specific integrated circuit). The switches internal to the FPGA architecture that provide reconfigurability also add parasitic capacitance and longer propagation paths. Both effects tend to reduce device speed. Therefore, optimum execution performance requires a balance between what is executed in software and what is executed in hardware. This implies that data must pass between the fixed processing elements and the reconfigurable media. With this arrangement, the execution of a synthesized structure requires three steps:

1. Input arguments must pass from the processor to the reconfigurable medium. The bus bandwidth connecting the two components governs the transfer time.
2. The reconfigurable medium executes the operation. In nonsequential

March 1993
structures, this would be the time it takes the inputs to propagate to the output(s).

(3) The output(s) of the operation are returned to the processor. State information, if generated, may stay within the reconfigurable resource.

A gain in performance is achieved only if these three steps take less than the average time needed to evaluate a given function in software. If required, the steps can be easily pipelined to operate concurrently with processor activities.

The PRISM-I prototype

PRISM-I is a proof-of-concept system that was developed to demonstrate the PRISM approach and, more generally, the viability of incorporating adaptation in a general-purpose machine.

**Configuration compiler.** We selected the C language for PRISM-I. C is far from an ideal hardware-description language, but it is widely accepted and used regularly for programming computationally intensive tasks.

Figure 2 illustrates the high-level procedure flow diagram detailing the PRISM-I configuration compilation. It shows the compiler accepting a C program as input and producing a hardware image and software image as output along with other intermediate data.

**Function identification/extraction.** The first step in the compilation process separates those portions of the program that are candidate hardware segments from those that will remain software segments. The prototype system supports this separation by imposing a minimum resolution of function subroutines on candidate segments. This segmentation granularity has distinct advantages. Most importantly, it simplifies system development. However, the programmer must be aware of this level of segmentation to properly structure applications for the architecture. Finer segmentations may yield better results and are currently under investigation.

The identification process focuses on candidates with the greatest impact on overall performance. This requires estimating the program's runtime behavior. Hardware-architecture and instruction-set models, probability models for conditional execution, and rule-based analysis can be used to predict this behavior without actually executing the program. There are some problems that limit the accuracy of these pro-profiling techniques—for example, coping with runtime data dependencies, and predicting IF-paths and

---

**Figure 2.** High-level block diagram illustrating the complete PRISM-I configuration compilation process.

---

**Figure 3.** Detail flow diagram for the hardware-image synthesis in the PRISM-I configuration compilation process.
loop iteration counts. In the PRISM-I prototype compiler, the identification of candidate functions requires some interaction. The compiler prompts the programmer with a list of synthesizable candidates, and the programmer indicates which ones to accept.

*Hardware-image synthesis.* Next, identified hardware candidate(s) are transformed into a physical description capable of reconfiguring the hardware platform. This step also produces an access-definition file, which specifies how the software image can access the newly synthesized functions.

Figure 3 presents a detailed flowchart of the hardware-image synthesis. The shaded area highlights components developed specifically for PRISM-I. The procedure is similar to the procedure for behavioral silicon compilation, and three of the components — the PRISM-I parser, dataflow graph generation, and reverse reduction equation extraction — have counterparts in conventional behavioral silicon compilers.

The process begins with a C function, which passes first through two standard Unix utilities, c and lint, which are used to verify the integrity of the input. Next, the PRISM-I parser converts the function description into an internal representation that can be easily manipulated. In addition, the parser produces software definitions that link subsequent function calls to the reconfigurable hardware. The parsed information is then used to build a dataflow graph (DFG) representation of the input function. (A DFG is a directed graph consisting of nodes and arcs. The nodes correspond to declarations and operators in the source code. The arcs directed toward nodes represent pathways for arguments to enter the operation, and departing arcs represent the computed result of the operation. For a sample function and its corresponding DFG, see the sidebar “Dataflow graph analysis,” page 16.)

After the DFG is generated, the compiler uses hardware structures from a structures library to instantiate behavioral elements into the DFG. A rule-based system transforms the high-level structural description into simplified Boolean equations.

The reduced Boolean expressions then pass to the PRISM-I partitioning/decomposition step. This step segments these expressions in a way that minimizes the reconfigurable resources required to implement the equation ensemble. An algorithm has been developed that provides high-quality partitioning in a reasonable amount of processing time. DFG information is also used at this point to predict common factors between the Boolean expressions.

The remaining tasks of hardware-image synthesis — netlist generation, and placement and routing — are target-device dependent. The netlist-generation block transforms the internal PRISM-I description into, in this case, a Xilinx netlist format, which then passes to the Xilinx place-and-route package. The output can then be used to configure one or more Xilinx FPGA devices.

*Access/merge redifinition.* After the successful generation of hardware images, the remainder of the program specification (that is, the portion not synthesized into hardware) is merged with the access-definition file to produce a new C program definition (see Figure 2). This step provides a means of telling the final compilation step how to efficiently access the newly created structures with minimal data movement. When the software image is loaded into memory in preparation for execution, the reconfigurable arrays must also be initialized and loaded with the appropriate hardware-image files. The configuration compiler inserts library function calls into the program stream during this step to handle loading and initialization.

*C compiler optimization.* The newly merged program specification is then compiled to form a software-image file. The software image created in this final step can be treated as any conventional executable file. When the software image is executed, the appropriate hardware images are loaded into the appropriate FPGA(s) automatically.

*Reconfigurable processing platform.* The PRISM-I hardware platform was our first attempt at a reconfigurable platform. The design was intentionally simple to demonstrate the utility of this architecture. It consisted of an existing processor board based on a 10-MHz Motorola 68010 processor (specifically, an Armstrong processing node) and a second board consisting primarily of four Xilinx 3090 FPGAs. The FPGAs support system reconfiguration in less than one second under software control. A 16-bit bus connects the two boards.

To evaluate synthesized operations, the processor explicitly moved the function input arguments to the appropriate FPGA bank and then immediately moved the computed result back to the processor board. This whole operation was accomplished with no wait states, yet it still required from 48 to 72 processor clock cycles to complete. Despite the gross inefficiencies in data movement, the prototype system exhibited surprisingly good performance.

**Experimental results**

Table 1 on page 17 summarizes the compilation and runtime performance for several functions encountered in many applications. While most of these functions are identifiable by the reader, useful target functions that would benefit common applications are likely to be more obscure. In these experiments, we used a Sun-3 optimizing C compiler to generate the M68010 code for trials without PRISM-I and to generate the software image for trials with PRISM-I.

The last column of Table 1 gives the speedup achieved by using PRISM-I versus conventional software evaluation for the listed functions. The table shows that executing a call to the synthesized Hamming function on the PRISM-I system requires 1/24th of the time required to evaluate the function entirely in software on the host processor. Note that the function Bitrev, a common bit-reversal routine used in fast Fourier transform algorithms, requires a For-loop and many logical and shifting operations when written in C (as in this case). The compiled Bitrev configuration requires no logic but merely a reassignment of the input pins to the outputs.

Table 1 shows that the compilation time for most of these examples is under 15 minutes (using a Sun Sparc IPC workstation). The short compilation time was an important design issue to ensure the PRISM concept's viability. The actual performance benefit for applications using these functions depends on how frequently the application references the synthesized structures. For example, the function LogicEv performs the core computation used in a digital circuit discrete-event simulator. If a digital-
Dataflow graph analysis

Figure B shows a sample C function, and Figure C shows its corresponding dataflow graph (DFG) presentation. The arcs directed toward nodes represent pathways for arguments to enter the operation; departing arcs represent the computed result of the operation. The sourceless primary input arcs entering the graph (A, B, and C from the top of Figure C) represent the values passed as inputs to the function. There is one loadless departing arc representing the computed function return value. To help show the connection between the DFG and the function description, the number preceding each line in Figure B appears in parentheses on the DFG in Figure C.

The DFG constructed in Figure C shows three types of nodes:

1. \textit{Function nodes}. These nodes denote operations that are members of \( C \)'s fundamental arithmetic and Boolean operational set (for example, +, -, AND, and OR). Operations in the source function map to these nodes in the graph.

2. \textit{Conditional nodes}. There is a conditional node for every If-statement in the source code. As shown here, this node represents a magnitude comparison operation between the two input arcs. The output arc represents the binary decision of the comparison (that is, True or False).

3. \textit{Multiplexer nodes}. There is a multiplexer (MUX) node for every conditional node. A MUX node, like a 2:1 multiplexer, has an output path equal to one of the two input paths. The output of the node is determined by a select line originating from a conditional node.

The DFG representation provides sufficient clues to synthesize the function in a purely modular fashion. If an assortment of functional blocks existed that could perform all the node operations listed above and also offer a means of interconnecting these blocks, then almost any arbitrary function could be constructed. However, such an implementation would likely be suboptimal in resource usage and dynamic performance because subsequent logic reduction steps could probably yield a better solution.

Other information can be derived from the DFG and applied to the synthesis process. Dangling branches, such as the declaration of \( y \) in Figure B (the shaded portion of Figure C), do not contribute to the function and can be pruned from the graph.

Information from the DFG is also used to predict common factors between the Boolean expressions generated in a subsequent step. These predictions are useful later in the partitioning process.

Figure B. Example code to illustrate DFG generation from a sample C function.

```c
short nothing(a, b, c) /* This function does nothing */
{
    char a, b, c;
    char x, y, z;
    /* Passed function variables. */
    /* Locally declared variables. */
    z = a + 5;
    x = z + b * 7;
    x = x + c;
    if ((x >> 1) > c) {
        x = x - c;
    }
    y = x ^ 7;
    if (b <= 0) {
        x = x ^ b;
    }
    return x + x;
}
```

Figure C. The corresponding DFG for Figure B, with the primary inputs entering at the top.

16
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description (input bytes/output bytes)</th>
<th>Compilation Time (min.)</th>
<th>Percent Utilization of XC3090 FPGA</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming(x, y)</td>
<td>Hamming metric calculation (4/2)</td>
<td>6</td>
<td>38</td>
<td>24</td>
</tr>
<tr>
<td>Bitrev(x)</td>
<td>Bit-reversal function (4/4)</td>
<td>2</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>Neuron(x, y)</td>
<td>Cascadable 4-input n-net function (4/4)</td>
<td>12</td>
<td>52</td>
<td>12</td>
</tr>
<tr>
<td>MultAccm(x, y)</td>
<td>Multiply/accumulate function (4/4)</td>
<td>11</td>
<td>58</td>
<td>2.9</td>
</tr>
<tr>
<td>LogicEv(x)</td>
<td>Logic-simulation engine function (4/4)</td>
<td>12</td>
<td>40</td>
<td>18</td>
</tr>
<tr>
<td>ECC(x, y)</td>
<td>Error-correction coder/decoder (3/2)</td>
<td>6</td>
<td>14</td>
<td>24</td>
</tr>
<tr>
<td>Find_first_1(x)</td>
<td>First “1” in input locater (4/1)</td>
<td>3</td>
<td>11</td>
<td>42</td>
</tr>
<tr>
<td>Piecewise(x)</td>
<td>Five-section piecewise linear segmenation (4/4)</td>
<td>24</td>
<td>77</td>
<td>5.1</td>
</tr>
<tr>
<td>ALog2(x)</td>
<td>Base-2 A*log(x) computation (4/4)</td>
<td>16</td>
<td>74</td>
<td>54</td>
</tr>
</tbody>
</table>

circuit-simulator application spent 90 percent of its execution time calling and executing LogicEv in software, it would perform 6.67 times faster with PRISM-I. The speedup factor in this case is derived by applying Amdahl’s Law: 6.67 = 18 / (0.9 + 18 - 0.9 * 18). The PRISM-I platform was hindered in its performance by the relatively slow interface between the processor core and the reconfigurable media. Newer microprocessors are capable of much higher bus transfer rates and can transfer data to/from the reconfigurable devices in significantly less time than the prototype system. We expect to curtail this problem (among others) by reducing the number of cycles required to access the reconfigurable media by a factor of more than 20. Accordingly, we expect an additional order-of-magnitude performance improvement.

Since PRISM-I is a proof-of-concept system, we did not incorporate certain functions into the compiler if they were not paramount to the compiler’s operation or if they had been demonstrated and documented elsewhere. Presently, the input source code to the configuration compiler is constrained by some major limitations:

- Neither state nor global variables are currently supported within the boundary of candidate functions. They must instead be passed through function arguments; however, local automatic variables can be declared freely.
- Limitations on the proof-of-concept system require the sizes of the input arguments to total no more than 32 bits. Likewise, the return value is limited to 32 bits.
- The exit condition for For-loops must be independent of the input arguments.
- Floating-point types and operations are not supported.
- Synthesized structures evaluate the function directly in a single cycle. They do not currently use sequential devices or feedback paths.
- Not all C constructs have been implemented. For example, the do-while and switch-case constructs are not supported in the prototype system.

These limitations, among others, offer a substantial challenge for future research in the development of a full-scale compiler.

We have presented an architecture and a high-level-language compiler that have the potential of significantly improving the execution performance of many applications. The PRISM approach adapts the configuration and fundamental operations of a core processing system to the computationally intensive portions of a targeted application. The viability of having compilers generate hardware configurations in conjunction with executable code means that the user can take advantage of the performance increase offered by application-specific hardware without having to become a hardware designer. The newly synthesized operations are targeted to RAM-based logic devices that provide a mechanism for fast processor reconfiguration.

The prototype system uses commercially available Xilinx FPGAs to facilitate reconfiguration. These are excellent devices for demonstrating PRISM principles. However, an FPGA device designed specifically to exploit such features of this architecture as shadow configurations, fixed input/output structure, and faster reconfiguration time may be essential for a PRISM system to support conventional operating system concepts like context switching, debugging, and resource sharing.

The PRISM-I prototype implementation, developed at Brown University’s Laboratory for Engineering Man/Machine Systems, is complete and operational. A full-scale version of the configuration compiler and hardware platform is under development.

References

3. P. Bertin, D. Roncin, and J. Vullemin, "Introduction to Programmable Active
The Department of Electrical Engineering and Computer Science is seeking applications for several tenure-track, visiting, and research positions in electrical engineering and computer science beginning Fall 1993. Candidates with research interests in communications, graphics and user-interfaces, software engineering, and data communications/computer networks are especially encouraged to apply. Other areas the department is interested in include artificial intelligence, databases, data networks, and computer security. Applications from other highly qualified individuals in other areas of electrical engineering and/or computer science are also sought. Visiting and research appointments are for up to one-year periods.

The Department of Electrical Engineering and Computer Science consists of more than 40 faculty members and several visiting professors. The department offers the BS, MS and Doctoral degrees. Over a thousand students are currently enrolled in all three degree programs. The School of Engineering and Applied Science, which includes the department, offers a wide variety of computer equipment that is accessible to students and faculty. The school also offers unprecedented opportunities for research cooperation with other departments in areas such as CAD-CAM, Operations Research, and Engineering Management.

The George Washington University is located in the heart of Washington, DC, only a few blocks west of the White House. Many major research funding agencies are located in close proximity to the campus and there are several other universities close by. A research and graduate teaching campus, set in a major research park in suburban Virginia near the Washington-Dulles Airport, opened in September 1991. Substantial equipment and student stipends are available for faculty capable of establishing government- and industry-supported research projects at the new campus.

Candidates should send a curriculum vitae, a research summary and names of at least three references as soon as possible but before the closing date of July 31, 1993, to:
Professor R.J. Harrington, Chairman
Department of Electrical Engineering and Computer Science
The George Washington University
Washington, DC 20052


Peter M. Athanas is an assistant professor in the Bradley Department of Electrical Engineering at Virginia Polytechnic Institute and State University. His research interests include reconfigurable computer architectures, logic synthesis, VLSI, and parallel processing. Athanas received a BSEE from the University of Toledo, an MSEE from Rensselaer Polytechnic Institute, and an ScM in applied mathematics and a PhD in electrical engineering from Brown University. He is a member of the IEEE Computer Society.

Harvey F. Silverman is dean of engineering at Brown University and director of the Laboratory for Engineering Man-Machine Systems. His current research interests include parallel architectures for signal processing and speech recognition, recognition and signal processing algorithms, acoustic systems for speech data acquisition, and real systems implementation issues.

Silverman received the BS and BSEE degrees from Trinity College in Hartford, Conn., and the ScM and PhD degrees from Brown University. He is a member of IEEE.

Readers can contact Peter M. Athanas at the Bradley Dept. of Electrical Engineering, Virginia Tech., 340 Whittmore Hall, Blacksburg, VA 24061.